A Transactional Model for Long-Running Activities

Umeshwar Dayal
Digital Equipment Corp.
1 Kendall Square, Bldg 700
Cambridge, MA 02139
dayal@crl.dec.com

Meichun Hsu
Digital Equipment Corp.
800 West El Camino Real
Mountain View, CA 94040
hsu@ocean.dec.com

Rivka Ladin
Digital Equipment Corp.
1 Kendall Square, Bldg 700
Cambridge, MA 02139
rivka@crl.dec.com

Abstract

Many computer-supported applications are of long duration and consist of multiple steps that are executed over possibly heterogeneous servers. Such activities have weaker atomicity requirements than transactions. Previously, we illustrated how to organize the execution of such activities using triggers and transactions. In this paper, we describe an execution model in which activities may consist recursively of steps that may be subactivities or transactions. The model defines precisely the semantics of activities: communication between steps and the failure semantics of activities including compensation and exception handling. The model also supports querying the status of activities. We also propose an implementation of the model using recoverable queues for reliably chaining the steps according to the semantics of the model.

1 Introduction

Many computer-supported applications are of long duration, and involve multiple steps of processing. The steps may be executed by different servers, perhaps on different nodes of a heterogeneous service network. For example, a purchase order may be issued from an inventory clerk, then passed to a manager who approves it, and then passed to an accountant who makes proper accounting entries. Because the steps of such an activity access shared, persistent data, they need to be synchronized among themselves and with the steps of other activities. Also, since the applications or the servers may fail, the failure semantics of activities need to be defined.

In conventional database management systems, the only unit of work supported is the transaction. However, activities have weaker concurrency and failure atomicity requirements than transactions. Executing a long-running activity as a single transaction is not strictly necessary in most cases, and can significantly delay the execution of short transactions. For example, if purchase order processing is run as a single transaction, locks on the inventory records and the budget records may be held for a long time, severely limiting database concurrency. When these steps involve several distributed servers, commit processing is also expensive, and the transaction can run only when all servers are available simultaneously. Moreover, in a heterogeneous system, some of the servers might not even be capable of participating in distributed commit processing.

One approach to handling long-running activities, therefore, is to have each step run as a transaction; thus, an activity consists of multiple transactions. In conventional transaction processing systems, the control flow among the steps is embedded in application programs (e.g. [McGe78]). There is no system support for handling failures or exceptions across the steps of an activity. Several extended transaction models to support activities have been proposed [GS87, KR88, Reut89, Garc90]. These models support declarative specification of control flow, and an automatic *compensation* capability that offers some level of failure atomicity for the activity. These models are all based on the conventional "flat" transaction model in which transactions are strictly sequential.

In this paper we describe ATM, a transactional model of activities that is based on an extended nested transaction model that we introduced in [HLM88, Chak89]. Like the original nested transaction model of [Lisk85, Moss81], our extended model is especially suitable for distributed systems because it supports intra-transaction parallelism by allowing a transaction to spawn *nested transactions* that execute concurrently. However, in the original nested transaction model, all the nested transactions are *immediate* in that they can be scheduled for execution as soon as they are spawned. Our extended model provides greater flexibility in specifying the scope of execution of a nested transaction: *deferred* nested transactions are executed at the end of a transaction; and *decoupled* nested transactions are executed concurrently with the spawning transaction.

Previously, we showed how to use this generalized transaction model and rules to organize and control long-running activities [DHL90]. Each step of an activity was modelled by a transaction. The control flow among the steps was expressed implicitly by rules. Thus, to start a step S2 after another step S1, we would write a rule that spawned a decoupled nested transaction to execute S2. The rule would be triggered either by S1 signalling an event or by the rule system detecting that some specified event had occurred in the database. Thus, in that model, an activity tacitly consisted of a single top transaction and all of the nested transactions (including decoupled transactions) spawned from it. We argued that the use of rules allowed the control flow to be dynamically modified based on the database state or the history of events that had occurred. Activities, thus, were not a first class concept in the model. For instance, activities could not be nested within other activities. The model did not include mechanisms for compensation or exception handling. Instead, rules were written to invoke alternate actions when exceptional conditions were detected.

In this paper, we develop ATM, a transactional activity model in which activities are treated explicitly as execution units in their own right. An activity specifies a computation structure that may consist recursively of other (sub)activities or of (top) transactions. (These transactions may contain immediate, deferred, and decoupled nested transactions.)

Control flow and data flow between the steps of an activity may be specified statically in the activity's *script*, or may be dynamically modified through the execution of rules triggered by events that occur as the activity progresses. While rules are not strictly necessary, they are still useful for checking constraints, triggering additional tasks, or modifying the flow in response to unanticipated conditions. We believe that scripts and rules provide a powerful combination of mechanisms for building activities, whose semantics are described by ATM.

The model defines precisely the semantics of activities, including compensation and exception handling. Upon failure, we allow activities to be aborted (committed steps are compensated; e.g., if the purchase order processing activity fails after the accounting step has already debited the account, a compensating step is to credit the account) or to handle the failure exception by executing an alternative step. Our model allows an activity to include steps that cannot be undone or compensated; we call these *critical* steps. Typically, these are steps that have external effects on the real world (e.g., mailing a cheque, firing a missile) and it is not desirable to allow their effects to become visible before the activity commits. An activity which includes critical steps is different from a transaction because some of this activity's steps may be non-critical, while all subtransactions of a transaction must be critical.

Additional features supported by the activity model are communication via parameter passing among the steps of an activity; and querying the status of an activity (i.e., whether its subactivities and transaction steps are active, committed, aborted, or compensated).

Our activity model can be layered on *any* model that supports atomic transactions, or even concurrently on more than one transaction model (e.g., in a heterogeneous system, different servers may support different transaction models). However, layering activities on top of our extended nested transaction model allows the use of the various types of nesting.

Section 2 provides a brief comparison with related work. Section 3 describes ATM. Section 4 gives an example application of ATM based on a hospital information system. Section 5 proposes an implementation architecture based on recoverable queues [BHM90].

2 Related Work

We classify related work into four categories. The first category includes early workflow•models for office or business procedures (e.g. [Zism78], [DZ81], [Barr82], [CC82], [LR83], [BP83], [WL86]). All these models included some notion of a task (sometimes called a procedure, action, or step). The flow of control between tasks was specified typically by augmented Petri nets or triggers. However, these early workflow models generally were non-transactional, and did not address the problems of data sharing, persistence, and failure recovery.

The category that is closest to our work includes various extended transaction models for long-running activities [GS87, KR88, Reut89, Garc90, ELLR90]. In the *saga* model of [GS87], an activity is a sequence of transactions $T_1, T_2, .., T_n$. After T_i is committed, T_{i+1} is invoked. If some T_k fails, then T_k is aborted and the system automatically invokes compensating transactions $C_{k-1}, ..., C_1$, in that order. The stictly sequential saga model is generalized in the *migrating transaction* model to allow concurrent execution of component transactions [KR88]. In addition, invariants on the database state that must be maintained to ensure the feasibility of running compensating transactions, can also be specified. In [Reut89], *contracts* are proposed as an extension to migrating transactions. The steps of a contract may be arbitrary sequential programs, not necessarily transactions. The control flow among the steps is specified as part of the contract definition. Context is maintained across the contract through the use of global variables. The multi-transaction activity model of [Garc90] uses mailboxes (persistent message queues) for control flow and data flow between steps of an activity. As in our earlier model [DHL90], an activity is initiated by a single step (transaction). Activities can be nested, and compensation can be provided for nested subactivites as well as for individual steps. The InterBase model of [ELLR90] was developed for heterogeneus database systems. A global transaction in their model, which corresponds to an activity in our model, consists of one or more steps (called subtransactions), each of which executes at a single database server. Their model allows the specification of alternative steps, of compensation steps, of non-compensatable (i.e., critical) steps, and of temporal constraints (which specify when steps are to be executed).

All of these models are based on conventional flat transactions; hence, they neither support nesting nor concurrency within a step. Our model, on the other hand, is based on the extended nested transaction model, and gives the activity designer a lot of flexibility to specify intra- and inter-step parallelism. Sagas, migrating transactions, and InterBase's global transactions are two-level structures (activities and transactions), whereas our model supports arbitrarily nested subactivities. Also, our model has richer failure semantics, supporting rollback (with compensation), roll forward, and alternative execution paths. Finally, the other models provide fixed control flow and a rigid compensation policy. In contrast, our model allows both the static specification of control flow (embedded in the activity's script) and its dynamic modification through rules (as we illustrated in [DHL90]). The execution semantics of rules are also described by the extended nested transaction model; hence, no extensions are necessary to incorporate rule execution into the activity model.

The multi-level transaction model also extends nested transactions [Weik86] by allowing a child action to commit independently of its parent. This aspect of the parent-child relationship is very similar to that between a parent activity and a child in our model. However, in the multi-level transaction model, the parent transaction is still serializable with its siblings at some level of data abstraction (i.e., the "commit" of the child action is in fact only a commit at a lower-level of abstraction). In our activity model, the parent activity is simply not required to preserve such atomicity.

The third category of related work includes several extended transaction models for long-running cooperative activities such as engineering design [KLMP84, KSUW85, PKH88, FZ89,Kais90, NZ90, KS90, RRD90] and text editing [EG89]. These models also support weaker notions of atomicity than the traditional transaction model. Their goal is to provide more sharing so that members of a design group can see one another's work in progress, while isolating one group from another. Our goal is somewhat orthogonal: to improve throughput by breaking up a long-running activity into short transactions.

Finally, there is work on generalized transaction frameworks, such as ACTA [CR90, Klei91]. This work is aimed at describing and comparing existing transaction models in terms of a small number of constraints. These frameworks as yet do not deal with the semantics of activities.

3 The Model

Our model consists of activities and transactions. An activity consists of multiple application steps each of which is either an activity or a transaction. Activities can be further nested. Thus, children of an activity may be activities or transactions or a combination of these.

Activities and transactions can be nested to arbitrary levels with the exception that activities cannot be created from within transactions. When our discussion is applicable to both transactions and activities, we refer to activities and transactions as ***actions***.

Nested actions form a tree. An action may contain any number of nested actions or ***subactions***, some of which may be performed sequentially, some concurrently. For convenience, we assume that there exists a distinguished system root, *Sys*, for activities and transactions. A ***top transaction*** is a transaction that is at the root of a transaction tree, i.e., a maximal tree that consists only of transactions. A ***top activity*** is an activity that is at the root of an activity tree, i.e., a maximal tree that consists of transactions and activities. An activity node is a super-root linking a forest of transaction trees into a single tree. Thus, below the activity node, subactivities and top transactions may exist.

We use standard tree terminology in referring to the relationship between actions, for example, ***parent***, ***child***, ***ancestor*** and ***descendant***.

In order to define the relationship between a child action and its parent action, let S be an action with P as its parent action. Then we can specify a ***child-parent*** relation between S and P based on whether or not S and P satisfy the following properties:

TERM : P commits only after S terminates.

CD : S is commit dependent on the commit of P.

SR : S is serializable with respect to P's other children.

VIS : S has access to all the objects that P has, e.g., it can read objects P has modified.

We assume that each ***activity class*** has a predefined ***activity description***, which describes the sequence of actions contained in the activities in this class. Each action item is described as either the name of an activity class, or the name of a top transaction class. The activity description also includes a specification of the *data flow* involved in the activity execution.

For the purpose of this discussion, we assume that a program which creates an activity is given a ***handle*** for the activity. After an activity is created, the program may ***query*** the status of the activity by presenting the activity handle to the system. The program may also ask the system to *cancel* the activity.

In the rest of this section we describe precisely the semantics of our ***Activities/Transactions Model***, or ATM for short. Our presentation consists of three parts: transactions, activities and failure handling. We start by describing the transaction part of the ATM. Then we present activities, and finally, we describe how we handle failures.

3.1 Transactions

Suppose that P and S are two transactions such that S is created by P. Then we say that P is the *creator* and S is the *createe.*

To give the programmer fine control over the scope in which a child transaction is executed, we allow the createe to be performed not as a child of its creator, but as a child of another parent which we denote as the child's *proper-parent.*

Let S be a nested transaction with P as its proper-parent. Then the ***child-parent*** relation between P and S satisfies the TERM, CD, SR and the VIS properites.

The CD condition indicates that if a subtransaction commits and its proper-parent aborts, the effects of the subtransaction will be undone. When a subtransaction S and all its ancestors up to, but not including, the top transaction commit, we say that S has *committed to the top.* When S's top transaction then commits we say that S has *committed through the top.* The top transaction commits only after all of its subtransactions have terminated

A subtransaction may be aborted without causing its proper-parent transaction to abort. Thus, upon the failure of its subtransaction, the proper-parent can either go on with other computation or create another subtransaction to retry the computation that was aborted.

Concurrency within a transaction is obtained by allowing the proper-parent to start concurrent subtransactions. While a child is running, its proper-parent is suspended. However, sibling subtransactions may execute concurrently. Siblings are serializable at each level of the transaction tree. Thus, there is no problem with concurrent siblings interfering with one another. Sequential siblings are ordered according to when they run. This structure can't be observed from the outside; i.e., the overall transaction still satisfies the atomicity properties.

In our model, the child-parent relation is always held with respect to the proper-parent. Between the createe and the creator, however, only a commit dependency specification is allowed. If such a commit dependency is specified, then the abort of a creator will cause the child created by it to be aborted.

Our model distinguishes three execution scopes: *immediate, deferred* and *decoupled.* If S's scope is *immediate*, then S is executed within P immediately upon its invocation, thus, its creator is also its proper-parent. (Note, that if all subtransactions execute in the immediate mode then our model is identical to the traditional nested transaction model.) If the mode is *deferred* or *decoupled*, then the creator is different from the proper-parent.

The execution of ***deferred subtransactions*** is explicitly delayed until the end of the user's top transaction T and before any deferred subtransaction is executed, a point we shall refer to as the ***cycle-0 end.*** Let P be the creator of S. Then, instead of executing S as a child of P, T is made S's proper-parent. The execution of S is explicitly delayed until T reaches cycle-0 end.

In addition to satisfying the child-parent relation with T, S is commit dependent on its creator, P. If more than one deferred subtransactions are created before T reaches its cycle-0 end, then all these subtransactions are started as concurrent subtransactions in *cycle 1* at cycle-0 end. If the processing of subtransactions in cycle 1 causes more deferred transactions to be created, the latter are started when all subtransactions in cycle 1 have finished, and are started as concurrent subtransactions of T in Cycle 2. The cycles of execution of T continue until the last cycle finishes in which no more deferred subtransactions are created. For example, in Figure 1, T_3 is a deferred subtransaction created by T_1.

A separate top transaction S can be created from inside another transaction. Such a ***"nested" top transaction*** is called a ***decoupled transaction.*** A decoupled top transaction will be represented by its own tree. In this case, S's proper-parent is T's proper-parent, which might be an activity or *Sys*.

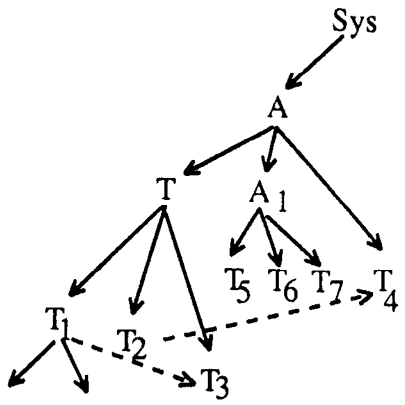
When S is decoupled, S can execute concurrently with T, and therefore, S might be serialized before T. This, however, may violate *causality*: T may see the results of S. Also, T may abort after S committed. Therefore, we allow the database programmer to specify whether the decoupled transaction is ***causally dependent.*** Let T be the top transaction and let S be the *causally-dependent transaction, CDtop* for short, created either by T or by one of its descendents. Then S is *causally dependent* on T iff S is serialized after T and S is commit dependent on its creator to commit through the top.

The execution of a *causally independent decoupled transaction* S has no special privileges relative to its creator T.

It is important to note that CDtop transactions whose commited creators have committed must be scheduled for execution. Therefore, CDtop transactions that are interrupted by a system failure should be automatically restarted as part of system recovery.

In Figure 1, a commit dependency is specified between T_4 and T_2, represented by the dotted arc between them. Therefore if T_2 aborts, then T_4 will be aborted. In Figure 1, the commit and serializability semantics of transactions in the solid tree (i.e., the proper-parent tree) is identical to that in a conventional nested transaction tree. In essence, the extension allows a nested transaction to be created from one spot in the tree and "grafted" somewhere else in

Figure 1: The Activity/Transactions Execution Tree



the tree, and it specifies the semantics of such "grafting".

So far, we have not restricted the serialization order among concurrent siblings. Sometimes, however, a particular serialization order might be desired. For this purpose, priorities can be assigned to transactions. The system guarantees that the serialization order of concurrent siblings of a proper-parent is consistent with their priority order.

The cycling mechanism interacts with the priority mechanism: within each cycle, the subtransactions are executed in priority order.

To constrain possible execution orders of concurrent CDtop transactions, we support a *pipelining* mechanism. We say that a decoupled transaction T' created by transaction T satisfies the pipelining property if for all transactions P that are serialized before (after) T, any decoupled transaction P' created by P is serialized before (respectively, after) T'.

3.2 Activities

Like a transaction, an activity can also be active, committed or aborted. The relationship between an activity and its children satisfies only the TERM and the VIS properties. Thus a a parent activity is committed only after all its children have terminated, and a child has access to all the objects that its parent has. However, the other two properties, CD and SR, are not satisfied. The commit of the children is independent of the commit of the parent activity. Therefore, if a parent activity is aborted, then all its *active* children are aborted; committed children, however, are compensated for. Sibling activities are *not* serializable; their effects on the database may be interleaved.

Aborting an activity is defined as follows. All children activities are aborted; all active top transactions are aborted. Committed top transactions *cannot* be aborted, their effects persist. Therefore, to support cancellation after an activity or a top transaction has committed, we provide an additional system facility which invokes compensation activities or transactions. When the abort of all the active parts of the activity has been completed, the compensation for committed transactions or activites is performed by executing the corresponding compensations in a an order that is the reverse of the orginal execution order.

The status of an activity can be derived from the *action tree* spawned by the activity. By preserving the action tree information in some form, the system can allow the users to query and display the status of the the different steps of the activity.

Assuming that some top transactions might be impossible to compensate for and hence should commit only if the parent activity commits, we allow the programmer to define whether a top transaction is *critical* or not. The commit of a critical transaction is only tentative and therefore its effects cannot be visible to its parent activity. To bound the duration in which critical transactions are tentatively committed we can specify whether activities are critical or not. All critical children of a critical activity stay tentatively committed; noncritical children commit independently of the fate of the parent activity. All critical children of a noncritical activity commit if the parent commits. This assumes that the parent can compensate at that level for their effects. For example, suppose that in Figure 1 the only critical actions are transaction T_5 and activity A_1. And suppose also that A aborts after A_1 has committed to it. Then in aborting A_1, A_1's compensation action needs to consider only the effects of T_6 and T_7 since T_5 is simply aborted. In summary, the actual commit of a critical action takes place when its parent commits through its *closest noncritical ancestor (CNA)*.

3.3 Exceptions

Requiring that every failure of a step cause an activity to be aborted is expensive, because rolling back an activity to its beginning could potentially undo a lot of work. As an alternative to aborting, our model supports exception handling. The goal is to allow nonfatal failed steps to be replaced by alternative steps, so that a transaction or activity can continue to make forward progress. One or more exception handlers can be associated with every (top) transaction, subtransaction, or subactivity, i.e., with every node in an activity tree.

When a child node, C, returns to its parent, P, with an exception condition (and aborts), the appropriate exception handler, E, (if one has been defined) is invoked. The exception handler is executed as a sibling of the failed node C. Note that E executes concurrently with any concurrent siblings of C that are still executing. However, sequential steps that are supposed to follow C are not initiated until exception handling terminates. E, in fact, can be thought of as performing an alternative task to that performed by C.

If E terminates successfully, then P can continue with its forward execution.

If E aborts or fails to perform the alternative task (in which case, too, it is aborted), or if no exception handler had been defined for C, then P has two ,options. The first option is that P itself aborts, possibly returning an exception condition up to *its* parent (thus, failure handling moves recursively up the activity tree). The second option is that P branches to an alternative computation path. Note that this requires no additional mechanisms. The conditional branches can be coded into P's logic, or implicitly invoked via rules.

The exception handling semantics described above are almost identical for activities and for transactions. The only difference is that if an activity node is to be aborted, then its already committed subactivities and top transactions must first be compensated for.

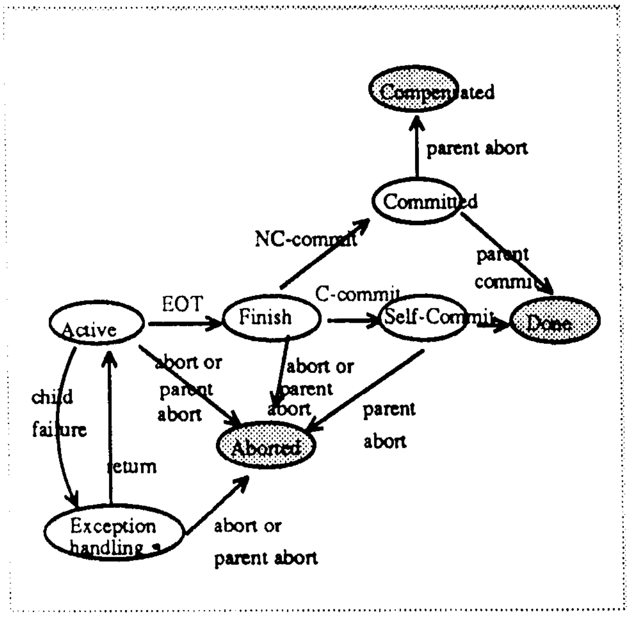
3.4 Summary

To summarize, we present in Figure 2, a state machine diagram that captures the behavior of our model. The multiple paths correspond to the executions of different types of actions. The diagram consists of nodes, arrows and labels: The nodes describe the different states in which an action might be, the arrows describe the legal transitions between states, and the labels on the arrows correspond to the conditions required by the respective transitions. The initial state is the *active* state and the final states are *done, aborted* and *compensated*.

We start by describing a failure-free execution. An action starts in the *active* state from which it exits by either reaching the end of its computation or a failure. Upon normal termination, the action enters the *finish* state. Note that an action stays in the *active* state until all its children terminate. Noncritical actions that commit move into the *committed* state; all critical actions move upon commit to the *self-commit* state. The *self-commit* state represents the state in which an action is only tentatively committed and can be simply aborted if its parent aborts. The *committed* state, on the other hand, corresponds to the state in which the effects of the action have already been committed to the "outside world", and therefore, the action can only be compensated for should its parent abort. When the effects of an action cannot be revoked by simply aborting it or compensating for it, the action moves into the *done* state. Noncritical actions move into the *done* state when their parent commits; critical actions move into this state when they commit through their CNA.

Next we consider non-failure-free executions. An abort of an uncommitted action (an action in the following states *active, finish, or exception-handling state*) moves it into the *aborted* state. The abort of the parent of an uncommitted or tentatively commited action will also cause the abort of the action and thus move it into the *aborted* state. As a result of an abort of the parent of a noncritical action that in is the *committed* state, the action moves into the *compensated* state.

Figure 2: Action State Diagram



Upon the failure of a child, the active action moves into an *exception-handling* state, in which it makes an attempt to accomplish the tasks of the failed child. It creates a child to perform the exception handling, and if the child terminates normally, it (the parent) moves back to the *active* state; otherwise, it aborts and therefore ends up in the *aborted* state.

4 An Example Application

This section outlines a patient information system for a hospital. This example is an adaptation of an example offered in [DHL90]. The example in [DHL90] focused on deferred and decoupled transactions spawned from database rules. In this section, we show how a mixture of the activity constructs and the database rules can be used to model the application. The database rules are expressed using the same syntax as that used in [DHL90]. The syntax used here for describing activities is self-explanatory.

The example models a long running activity that starts when a patient arrives at the hospital, continues through stages of examination and tests, until the patient is discharged from the hospital. For brevity we omit many details. However, through the simple example, we illustrate sequential and parallel control flow of activities and transactions, nested subactivities and subtransactions, simple data flow, critical actions, exception handling, and deferred and decoupled CD_top subtransactions spawned by database rules.

As in [DHL90], we assume that all pertinent information about the patient is recorded in a database that is shared by all the organizations involved. When a patient arrives, an event containing the patient's social security number is signaled which causes the Treat_Patient activity to be invoked. Treat_Patient in turn invokes subactions Admit, Notify_Doctor, Examine, Test and Discharge. The relevant activity descriptions are shown in figure 3.

Admit itself is an activity and consists of two subactions, Create_Adm_Record and Assign_Doctor. Create_Adm_Record returns an admission number AdNo. AdNo is sufficient to locate the patient's folder in the database. Assign_Doctor is itself an activity, consisting of a critical subaction, Schedule_Doctor, and another subaction Confirm. Confirm consults the patient to confirm the doctor assignment. If Confirm fails, the activity Assign_Doctor is aborted. Since Schedule_Doctor is a critical subaction, abort of Assign_Doctor will automatically cause it to be aborted. Abort of Assign_Doctor is reported as a failure signal to its parent Admit which in turn "aborts" itself by applying a compensation subaction Cancel_Adm_Record. Abort of Admit is reported to its parent Treat_Patient which simply terminates.

If the patient is admitted successfully, s/he goes through an examination-test loop. For examination, the doctor is notified, and when the doctor acknowledges (by signaling Doctor_Ack), the Examine action takes place. If tests are prescribed as a result of Examine, then the Test activity is invoked, and when Test is done, the doctor is notified again as part of the examine-test loop. Otherwise, the patient exits the examination-test loop and is discharged.

The Examine action is executed as an (interactive) transaction. The doctor prescribes medications and tests during this transaction. A database rule, Check_Conflicts, is triggered whenever prescriptions are inserted into a patient's folder to check for conflicts of medications and tests. It is spawned as a deferred subtransaction within the Examine transaction. The rule is shown in figure 3.

The Test action is an activity that consists of Schedule_Test followed by parallel performance of Schedule_Lab, Notify_Lab, and Execute_Test subactions for each test subscribed. When all tests are done, the Test activity terminates, and the doctor is notified. However, if any test reveals life-threatening condition, a database rule High_Priority_Notification is triggered which immediately notifies the doctor without waiting for other tests to finish. This rule is triggered as a causally dependent decoupled top transaction (CD_top) spawned from the Execute_Test transaction. This rule is shown in figure 3.

```
Activity Treat_Patient (SSNo)
  Admit (SSNo,AdNo,Need_test)
  repeat until not Need_test
    Notify_Doctor (AdNo)
    ON Doctor_Ack(AdNo)
    DO
      Examine (AdNo, Need_test)
      if Need_test
        Test (AdNo,Test_list)
    endrepeat
  Discharge(AdNo)
  On Failure(Admit)  /* exception */
     Terminate(Signal Failure)
  End Activity

Activity Admit (SSNo,AdNo,Need_test)
  Create_Adm_Record (SSNo,AdNo)
  Assign_Doctor (AdNo,Active)
  ON Failure(Assign_Doctor) /* exception */
    DO
      Cancel_Adm_Record (AdNo)
      Terminate(Signal Failure)
  End Activity

Activity Assign_Doctor(AdNo,Active)
  critical Schedule_Doctor(AdNo)
  Confirm(AdNo,Active)
  ON Failure(Confirm)
    Abort(Signal Failure)
  End Activity

Activity Test (AdNo)
  Schedule_Test (AdNo,Test_list)
  parfor T in Test_list
    Schedule_Lab(AdNo,T)
    Notify_Lab(AdNo,T)
    ON Lab_Ack(AdNo,T)
      Execute_Test(AdNo,T)
    endparfor
  End Activity

Rule Check_Conflicts
ON  {insert Prescriptions (AdNo)}* ; EOT
DO
  begin_transaction
    presc_check (AdNo);
  end_transaction
  begin_transaction
    notify_doctor(AdNo,D);
    get_input (D, op);
    if op = ''Abort'' then abort_top;
      else: execute op;
  end_transaction

Rule High_Priority_Notification
ON insert Procedure_Result (AdNo)
DO
  begin_CDtop
    if Dangerous(AdNo)
    then
      begin_subtransaction (priority high)
        notify_doctor(AdNo);
      end_subtransaction
  end_CDtop
```

Figure 3: Activity and Rule Descriptions

5 A Proposed Implementation Architecture

In this section we propose a simple implementation of the ATM model. The implementation described in this section is to illustrate *one* particular way of materializing the model. It is not intended to address efficiency or optimality tradeoffs among several possible implementations. In particular, we have opted for a design which builds on top of two other service abstractions which are either already available or well understood.

The proposed implementation assumes that a simple nested transaction service [Lisk85. Moss81] is available. Additionally, it uses services of a *reliable queueing facility*. The queueing facility is based on the queue abstraction as described in [BHM90], with additional primitives to allow for dynamic creation of queues. In essence, the nested transaction implementation constitutes the backbone of the system that offers atomic and persistent computations. The reliable queueing facility, which itself relies on the availability of an underlying transaction system, offers the ability for the system to *connect* the atomic computations together in a reliable and persistent manner. Together, they enable us to devise a simple implementation for our activity/transaction model. For brevity, we do not consider critical actions in this proposed implementation.

5.1 Overview

Activities execute as a sequence of top transactions or a sequence of concurrent blocks of top transactions. Every top transaction creates an *input queue* and is given a handle to an *output queue*. The input queue is used to capture the original top transaction request as well as deferred subactions requested during execution of the transaction. The output queue is used to synchronize with, or communicate the result to, subsequent top transactions in an activity.

A top transaction executes a sequence of subtransactions. Each subtransaction removes an element from the input queue and performs the work described by the element. A subtransaction may request deferred actions by inserting the request into the input queue of the top transaction. A subtransaction may also request a decoupled action by inserting the request into a *system queue* SYSQ. Every element in SYSQ will cause a top transaction to be spawned by the system. A top transaction finishes when its input queue is empty (i.e., when all deferred subtransactions are executed). Upon finishing, based on its "activity context" (to be explained later) it may create and insert additional queue elements into SYSQ to generate subsequent top transactions that belong in the activity that it is embedded in.

All operations on queues participate in nested transaction semantics. By the use of transactional queues, subactivities and decoupled transactions are reliably chained together to ensure persistent progress. Upon system recovery, the queue elements in SYSQ are automatically recovered, and the processing continues. The status of a running activity can be determined by recursively tracing through the input and output queues of the top transactions involved in the activity.

5.2 The Reliable Queuing Services

A *reliable queue* is an abstract data type that stores *queue elements*. The component of the system which manages queues and execute operations on queues is called a *queue manager*. Operations on queues are in general issued from within transactions. Queue managers therefore participate in transaction execution as conventional database managers.

Reliable queues have been described in [BHM90] as a vehicle for implementing reliable request processing in a transaction processing system where the *client* may use the queue operations to reliably capture its (simple) state and to recover properly from failures. We briefly summarize the queue operations below.

Unless otherwise specified, every queue operation is transactional: its effect persists if and only if the invoking transaction commits. We first define the following operations for creating and destroying queues:

- *q = create_queue()*: creates a queue and returns a queue handle *q* for the queue.
- *destroy_queue(q)*: destroys the queue designated by the queue handle *q*.

Queues are accessed using the following data manipulation operations:[1]

- *enq(q,qe)*: creates an element *qe* and stores it in a queue *q*.
- *enq_immediate(q,qe)*: creates an element *qe* and stores it in a queue *q*. Its effect is visible immediately, regardless of whether the invoking transaction commits.
- *qe = deq(q)*: deletes an element *qe* from a given queue *q*, and returns it to the caller. If the invoking transaction aborts, then the element is marked with an *abort code* and returned either to the given queue or to a separate error queue (which can be specified by a parameter in the call). If *q* is empty, the invoking transaction is suspended until an element arrives. A queueing discipline can be specified by associating a field of a queue element as the priority. The Dequeue operation will dequeue the next item with the smallest priority value.

[1] Semantics of these operations are adapted from [BHM90]

```
TSP(inq,outq):
  Begin Trx;  /* executes as a single top trx */
  qe = deq(SYSQ); /* qe contains sync & other info*/
  inq = create_queue(); /* need an inq */
  outq = qe.outq; /* extracts outq handle from qe */
  for each sq in qe.sync /* if sync needed */
    sqe = deq(sq); /* sync with other top trx*/
    endfor;
  enq(inq, qe.trx); /* qe.trx is first subtrx*/
  current_cycle = 0;
  Do while inq not empty;
     Begin_Trx; /* next subtrx */
     sub_qe = deq(inq);
     if sub_qe.cycle > current_cycle then
       current_cycle = sub_qe.cycle; /*inc*/
     execute sub_qe.task;
     End_Trx;
  /* finishing */
  if qe.output_needed then
    enq(outq, return_value);
  if qe.has_context then /*if is in an activ.context*/
    for each next_t in qe.context {
      cq = find_appropriate_out_queue();
      nextqe = package();  enq(SYSQ,nextqe);}
  destroy_queue(inq); /* suicide */
  End Trx;  /* end top transaction */
```

Figure 4: Pseudo-code for TSP

5.3 Transaction Service Program

A top transaction is created when a thread is dispatched to service a queue element in SYSQ. The thread executes the transaction service program (TSP). The pseudo-code for a simple TSP is shown in figure 4.

TSP removes a queue element qe from SYSQ and creates an input queue inq for itself. The element qe contains information about the task the top transaction is to perform, as well as other relevant information needed, including its activity context, requirement to synchronize with other top transactions, and an output queue handle. We omit the details of the structure of these information. TSP extracts the task (qe.trx) to be performed from qe and inserts it into its own input queue. It then starts a loop of subtransactions to service its input queue.

To implement deferred actions of a transaction, the system translates begin_deferred_action calls into enqueue operations on the input queue of the top transaction. To implement the cycle discipline, TSP carries a variable for "current cycle". When a deferred action is requested, the run-time checks this number and encodes the next higher number as the priority of the queue element representing this deferred action. This cycle number is incremented when the top transaction server detects the next element dequeued to be of a higher cycle number.

To implement decoupled actions of a top transaction, the system translates a begin_decoupled_top or begin_CD_TOP call into an enqueue or enqueue_immediate operation on SYSQ. If it is a CD top, then the enqueue operation is used. If it is an independent top transaction, then the enqueue_immediate operation is used.

It is assumed that the activity context is passed to a top transaction T through the queue element qe that T obtained from SYSQ (qe.context). If this context variable indicates that the top transaction is executed in the context of an activity, then, upon finishing (i.e., upon exhausting its deferred subtransactions), T packages and inserts a queue element in SYSQ in order to cause a top transaction T1 to be spawned for the activity. It also creates an additional output queue for T1, or passes its own output queue handle along to T1. To ensure that the decoupled top transactions spawned by T are executed before T1, the queue element communicated by T to T1 optionally contains a list of the output queue handles (qe.sync) of the decoupled transactions of T. T1 waits for termination of these decoupled transactions by dequeueing from these queues.

The parallel activity model can be implemented by creating multiple subsequent top transactions upon finishing, and by creating an appropriate *join* top transactions which waits for termination of the parallel ones.

6 Conclusion

This paper addresses the problems of reliable control flow management for long-running activities. Such activities have weaker atomicity requirements than transactions. The contributions of the paper are the definition of ATM, a rich transactional activity model, and a proposed implementation of the model.

In ATM, an activity consists of one or more steps, each of which is itself an activity or a transaction. The model supports communication between the steps of an activity; the failure semantics of activities, including compensation and exception handling (execution of alternative steps); and querying the status of an activity. We believe that scripts and rules provide a powerful combination of mechanisms for building activities, whose semantics are described by ATM.

In a previous paper [DHL90], we had illustrated the use of rules alone to implicitly chain the steps of an activity. We had argued that rules provided dynamic flow. However, the use of rules alone makes it more difficult to comprehend the computations performed by an activity. The ATM model introduced in this paper directly adds the concept of activities and their semantics to the extended nested transaction model. This has two benefits. First, it provides richer structure for activities, since they can now consist recursively of subactivities in addition to transactions. Second, it allows control and data flow in an activity to be explicitly defined via a script, without requiring rules. Rules are still useful for checking constraints, triggering additional tasks, or modifying the flow in response to unanticipated conditions.

The proposed implementation of ATM is modular. It relies on the implementation of two abstractions: nested transactions and recoverable queues. Additional work is needed to provide an implementation for critical actions.

Ideally, scripts and rules should be equally expressive for defining the conditions that control the invocation of steps. Our rule model, Defining an expressive language for scripts and rules is the subject of future work.

Acknowledgments

We wish to thank Adel Ghomeny and Charly Kleissner at Digital TPwest for inspiring discussions on the notion of an activity description, and Phil Bernstein for many valuabel suggestions.

References

[BHM90] Bernstein, P.A., M. Hsu, B. Mann, "Implementing Recoverable Requests using Queues." *Proc. ACM SIGMOD Conf.*, May 1990.

[Chak89] Chakravarthy, S., et al., "HiPAC: A Research Project in Active Time-Constrained Database Management. Final Technical Report." Xerox Advanced Information Technology, Cambridge, Mass., July 1989.

[CR90] Chrysanthis, P.K., K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior." *Proc. ACM SIGMOD Conf.*, May 1990.

[Daya88] Dayal, U., "Active Database Systems." *Proc. 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[DBM88] Dayal, U., A. Buchmann, D. McCarthy, "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System", *Proc. 2nd International Workshop on Object-Oriented Database Systems*, West Germany, September 1988.

[DHL90] Dayal, U., M. Hsu, R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions." *Proc. ACM SIGMOD Conf.*, May 1990.

[EG89] Ellis, C.A., S.J. Gibbs. "Concurrency Control in groupware Systems." *Proc. ACM SIGMOD Conf.*, June 1989.

[ELLR90] Elmagarmid, A.K., Y. Leu, W. Litwin, M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase." *Proc. VLDB Conf.* , August 1990.

[Garc90] Garcia-Molina, H., et al., "Coordinating Multi-Transaction Activities." Report UMIACS-TR-90-24, CS-TR-2412, Computer Science Technical Report Series, University of Maryland, College Park, MD.

[GS87] Garcia-Molina, H. and K. Salem, "Sagas," *Proc. ACM SIGMOD Conf.*, May 1987.

[HC88] Hsu, M. and T.E. Cheatham, "Rule Execution in CPLEX", *Proc. 2nd International Workshop on Object Oriented Database Systems*, West Germany, September 1988.

[HLM88] Hsu, M., R. Ladin, and D. McCarthy, "An Execution Model for Active Database Management System," *Proc. 3rd International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[Kais90] Kaiser, G.E., "A Flexible Transaction Model for Software Engineering." *Proc. IEEE Conf. on Data Engineering*, Feb. 1990.

[KLMP84] W. Kim, R. Lorie, D. McNabb, W. Plouffe, "A Transaction Mechanism for Engineering Design Databases." *Proc. VLDB Conf.*, August 1984.

[KSUW85] "Klahold, P., G. Schlageter, R. Unland, W. Wilkes, "A Transaction Model Supporting Complex Applications in Integrated Information Systems." *Proc. ACM SIGMOD Conf.*, May 1985.

[Klei91] Klein, J. "Advanced Rule Driven Transaction Management." *Proc. IEEE COMPCON Spring 1991.*

[KS90] Korth, H.F., G.D. Speegle, "Long Duration Transaction in Software Design Projects." *Proc. IEEE Conf. on Data Engineering*, Feb. 1990.

[KR88] Klein, J. and A. Reuter, "Migrating Transactions," *Future Trends in Distributed Computer Systems in the '90s*, Hong Kong, 1988.

[Lisk85] B. H. Liskov. "The Argus Language and System." *Distributed Systems: Methods and Tools for Specification*. pp. 343-430. Springer-Verlag, Berlin 1985.

[McGe77] McGee, W.C., "The Information Management System IMS/VS Part V: Transaction Processing Facilities," *IBM Sys. Journal*, Vol. 16, No 2., 1977, pp. 148-169.

[Moss81] J. Moss. "Nested Transactions: An Approach To Reliable Distributed Computing." MIT Laboratory for Computer Science, MIT/LCS/TR-260 1981.

[NZ90] Nodine, M.H., S.B. Zdonik. "Cooperative Transaction Hierarchies: A Transaction Modle to Support Design Applciations." *Proc. VLDB Conf.*, Aug. 1990.

[RRD90] Rauft, M.A., S. Rehm, K.R. Dittrich. "How to Share Work on Shared Objects in Design Databases." *Proc. IEEE Conf. on Data Engineering*, Feb. 1990.

[Reut89] Reuter, A., "Contracts: A Means for Extending Control Beyond Transaction Boundaries," Presentation at 3rd Workshop on High Performance Transaction Systems, Pacific Grove, CA, September 1989

[Weik86] "A Theoretical Foundation of Multi-Level Concurrency Control", *PODS Proceedings*, 1986.